

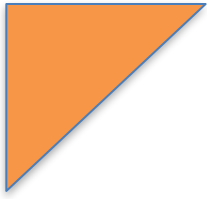
Monash University

ECE3073 Computer Systems

L02-1 NIOS

Clive Maynard

clive.maynard@monash.edu



COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of Monash University pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.



L2 - Nios Assembler & Machine code

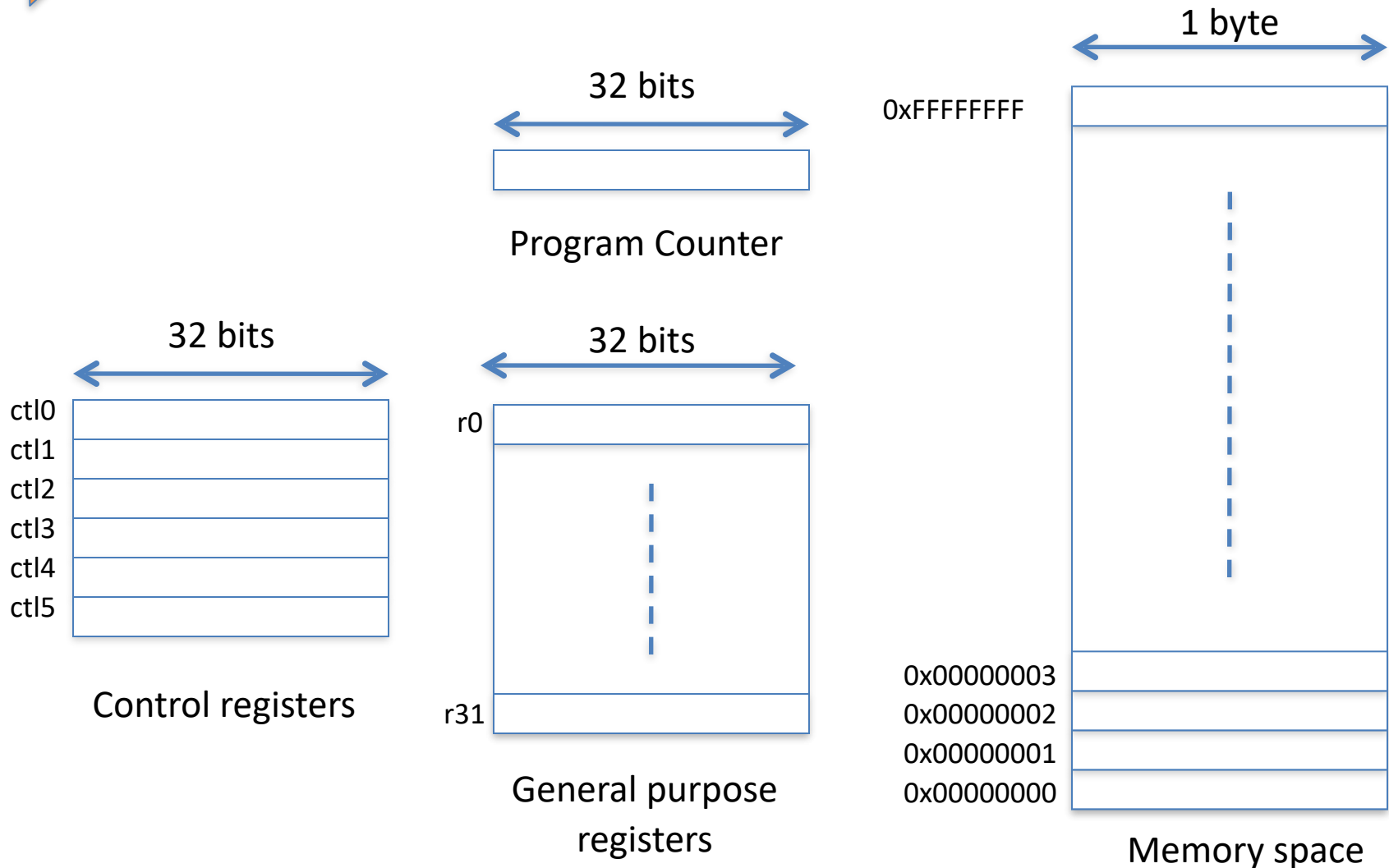
The purpose of this section is to become familiar with the Nios II machine code and assembler. You should note the similarity between MIPS (which some of you studied in ECE2071) and NIOS

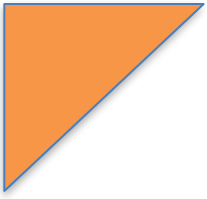
Although high level languages such as C can be used for many programming tasks, there are some which can only be performed in assembler. For instance C code cannot run until the stack has been set up – by assembly code. You will also be looking at program latency which can only be calculated by looking at service code at the assembler/machine code level.

The information given in this section will allow you to be able to translate between assembler and machine code

You will also be able to write and understand code in Nios assembler.

Nios II Programmer's Model





Memory

- byte addressed (2^{32} address space \sim 4GBytes)
- As with the MIPS processor, instructions must be on a 4-byte boundary.
- Registers for controlling input/output systems are memory mapped. They must be directly addressed (not accessed via the cache).



General Purpose Registers

Register	Name	Function
r0	zero	0x00000000
r1	at	Assembler Temporary
r2		
r3		
.	.	.
.	.	.
.	.	.
r23		
r24	et	Exception Temporary (1)
r25	bt	Breakpoint Temporary (2)
r26	gp	Global Pointer
r27	sp	Stack Pointer
r28	fp	Frame Pointer
r29	ea	Exception Return Address (1)
r30	ba	Breakpoint Return Address (2)
r31	ra	Return Address
(1) The register is not available in User mode		
(2) The register is used exclusively by the JTAG Debug module		

General Purpose is a misnomer here:

r0 is zero

r1 cannot be used

r24 to r31 have specified uses.

From “Introduction to the Altera Nios II Soft Processor”

Figure 2. General Purpose registers.

Control Registers

Register	Name	b_{31} ... b_2	b_1	b_0
ctl0	status	Reserved	U	PIE
ctl1	estatus	Reserved	EU	EPIE
ctl2	bstatus	Reserved	BU	BPIE
ctl3	ienable	Interrupt-enable bits		
ctl4	ipending	Pending-interrupt bits		
ctl5	cpuid	Unique processor identifier		

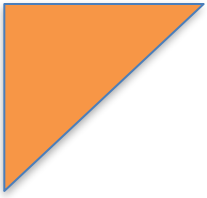
From “Introduction to the Altera Nios II Soft Processor”

Figure 3. Control registers.

Control registers accessed using rdctl and wrctl instructions.
We will be mainly interested in ctl0:

Register *ctl0* reflects the operating status of the processor. Only two bits of this register are meaningful:

- *U* is the User/Supervisor mode bit; $U = 1$ for User mode, while $U = 0$ for Supervisor mode.
- *PIE* is the processor interrupt-enable bit. When $PIE = 1$, the processor may accept external interrupts. When $PIE = 0$, the processor ignores external interrupts.



Program Counter

A 32 bit register (in reality a 30 bit register **plus** two least significant bits set to 0) which contains the address of the next instruction to be executed.

Usually updated when an instruction is fetched and before execution so if this value is referenced within an instruction it has already been modified.



Instruction Set

Most digital computers will support instructions from the following groups:

- **Data transfer:**

- Between registers **mov**, immediate data to register **movi** and **movui** and an immediate address **movia**.
- Between registers and memory (load **ld** and store **st**, word **w**, halfword **h** and byte **b**, signed or unsigned **u**, an address **a**, there is a special case for I/O devices which bypasses the cache **io**)

- **Arithmetic** (**add**, **addi**, **sub**, **subi**, **mul**, **muli**, **div** and **divu**)

- **Logical and shift/rotate** (**and**, **or** and **xor**, between registers or immediate 16-bit data **i** or 16 high-order bits with low order bits all zero **hi**)(shift **sr**, **sl**, arithmetic **a**, logical **l**, immediate data **i**)(rotate **ror**, **rol**, immediate **i**)



More Instruction Set

- Comparison

`cmpeqr`, `cmpner`, `cmpger`, `cmpgeur`, `cmpgtr` (pseudo-op), `cmpgtur` (pseudo-op),
`cmpler` (pseudo-op), `cmpleur` (pseudo-op)

Immediate **i** versions of the comparison instructions involve an immediate operand

- Branch and Jump instructions

Unconditional `jmp` rA and `br` LABEL

and conditional `beqr`, `bner`, `bger`, `bgeur`, `bgtr` (pseudo-op), `bgtur` (pseudo-op),
`bler` (pseudo-op), `bleu` (pseudo-op)

- Subroutine linkage

`call` LABEL, `callr` rA, `ret`

- Control Instructions

`rdctl`, `wrctl` read and write control registers

`trap`, `eret` similar to call and return, they are used for exceptions

`break`, `bret` used for software debugging

Exception Handling

What is an EXCEPTION?

- Software Trap
- Hardware Interrupt
- Unimplemented Instruction

Register	Name	b_{31} ... b_2	b_1	b_0
ctl0	status	Reserved	U	PIE
ctl1	estatus	Reserved	EU	EPIE
ctl2	bstatus	Reserved	BU	BPIE
ctl3	ienable	Interrupt-enable bits		
ctl4	ipending	Pending-interrupt bits		
ctl5	cpuid	Unique processor identifier		

Figure 3. Control registers.

In response the Nios II processor performs the following actions:

1. Saves the existing processor status information by copying the contents of the status register(ctl0) into the estatus register(ctl1)
2. Clears (changes to zero) the Ubit in the status register, to ensure that the processor is in the **Supervisor mode**
3. Clears (changes to zero) the PIE bit in the status register, thus **disabling any additional external processor interrupts**
4. Writes the address of the instruction after the exception into the ea register (r29 - exception return address register)
5. Transfers execution to the address of the exception handler (default address 0x00000020) which **determines the cause of the exception** and **activates an appropriate exception routine** to respond to the exception



Software Trap

A software exception occurs when a **trap** instruction (trap OP = 0x3A, OPX = 0x2D) is encountered in a program.

This instruction saves the address of the next instruction in the **ea** register (r29).

Then, it disables interrupts and transfers execution to the exception handler.

In the exception-service routine the last instruction is **eret** (Exception Return), which returns execution control to the instruction that follows the **trap** instruction that caused the exception.

The return address is given by the contents of register **ea**.

The **eret** instruction restores the previous status of the processor by copying the contents of the **estatus** register into the status register.

A common use of the software trap is to transfer control to a different program, such as an operating system.



Hardware Interrupts

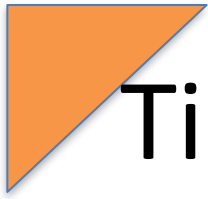
Hardware interrupts can be raised by external sources, such as I/O devices, by asserting one of the processor's 32 interrupt-request inputs, **irq0** through **irq31**. An interrupt is generated only if the following three conditions are true:

- The **PIE** bit in the status register is **set to 1**
- An interrupt-request input, **irqk**, is **asserted**
- The corresponding interrupt-enable bit, **ctl3k**, is **set to 1**

The contents of the **ipending** register (**ctl4**) indicate which interrupt requests are pending. An exception routine determines which of the pending interrupts has the highest priority, and transfers control to the corresponding interrupt-service routine.

Register	Name	b_{31} ... b_2	b_1	b_0
ctl0	status	Reserved	U	PIE
ctl1	estatus	Reserved	EU	EPIE
ctl2	bstatus	Reserved	BU	BPIE
ctl3	ienable	Interrupt-enable bits Pending-interrupt bits Unique processor identifier		
ctl4	ipending			
ctl5	cpuid			

Figure 3. Control registers.

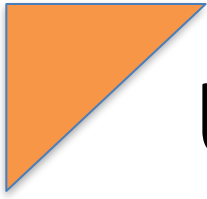


Tidying up After H/W Interrupts

Upon completion of the interrupt-service routine, the execution control is returned to the interrupted program by means of the `eret` instruction, as explained above.

However, since an external interrupt request is handled without first completing the instruction that is being executed when the interrupt occurs, the interrupted instruction must be re-executed upon return from the interrupt-service routine.

To achieve this, the interrupt-service routine has to adjust the contents of the `ea` register which are at this time pointing to the next instruction of the interrupted program. Hence, the value in the `ea` register has to be decremented by 4 prior to executing the `eret` instruction.



Unimplemented Instructions

This exception occurs when the processor encounters a **valid instruction** that is **not implemented in hardware**. This may be the case with instructions such as **mul** and **div**. The exception handler may call a routine that emulates the required operation in software.



Dealing with an Exception

When an exception occurs, the exception-handling routine has to determine what type of exception has occurred. The order in which the exceptions should be checked is:

1. Read the ipending register to see if a hardware interrupt has occurred; if so, then go to the appropriate interrupt-service routine.
2. Read the instruction that was being executed when the exception occurred. The address of this instruction is the value in the ea register minus 4. If this is the trap instruction, then go to the software-trap-handling routine.
3. Otherwise, the exception is due to an unimplemented instruction.

Machine Code Instruction Types

Table 8–1. I-Type Instruction Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A					B					IMM16																		OP			

- I-type – Five-bit fields A and B are used to specify general purpose registers.
A 16-bit field IMM16 provides immediate data which can be sign extended to provide a 32-bit operand.

Machine Code Instruction Types

Table 8–2. R-Type Instruction Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A					B					C					OPX											OP					

- R-type – Five-bit fields A, B and C are used to specify general purpose registers.
An 11-bit field OPX is used to extend the OP code.
Of these only the 6bit field msbits are used for Opcode information.

Machine Code Instruction Types

Table 8–3. J-Type Instruction Format

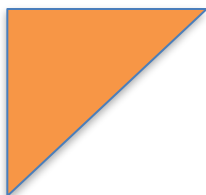
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMM26																										OP					

- J-type – A 26-bit field IMM26 contains an unsigned immediate value. This format is used only in the Call instruction.



Encoding addresses

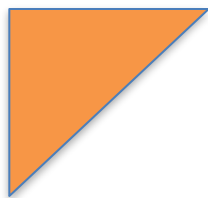
- For I-type instructions such as the branch instructions IMM16 is treated as a **signed number of bytes** relative to the instruction immediately following the branch.
- For J-type instructions such as **jmp** the IMM26 field is treated as an **absolute word address**. To create a fully 32-bit address, two zeros are attached in the two least significant bit (LSB) positions, followed by IMM26 and the final bits 31 to 28 are copied from the current program counter (PC) contents.
- To jump to any location in memory a computed jump **jmp** rA is used where register rA contains a full 32-bit address.



OP Encoding

Table 8–1. OP Encodings

OP	Instruction	OP	Instruction	OP	Instruction	OP	Instruction
0x00	call	0x10	cmplti	0x20	cmpeqi	0x30	cmpltui
0x01	jmp	0x11		0x21		0x31	
0x02		0x12		0x22		0x32	custom
0x03	ldbu	0x13	initda	0x23	ldbuio	0x33	initd
0x04	addi	0x14	ori	0x24	muli	0x34	orhi
0x05	stb	0x15	stw	0x25	stbio	0x35	stwio
0x06	br	0x16	blt	0x26	beq	0x36	bltu
0x07	ldb	0x17	ldw	0x27	ldbio	0x37	ldwio
0x08	cmpgei	0x18	cmpnei	0x28	cmpgeui	0x38	rdprs
0x09		0x19		0x29		0x39	
0x0A		0x1A		0x2A		0x3A	R-type
0x0B	ldhu	0x1B	flushda	0x2B	ldhuio	0x3B	flushd
0x0C	andi	0x1C	xori	0x2C	andhi	0x3C	xorhi
0x0D	sth	0x1D		0x2D	sthio	0x3D	
0x0E	bge	0x1E	bne	0x2E	bgeu	0x3E	
0x0F	ldh	0x1F		0x2F	ldhio	0x3F	



OPX Encoding

Table 8–2. OPX Encodings for R-Type Instructions (Part 1 of 2)

OPX	Instruction	OPX	Instruction	OPX	Instruction	OPX	Instruction
0x00		0x10	cmplt	0x20	cmpeq	0x30	cmpltu
0x01	eret	0x11		0x21		0x31	add
0x02	roli	0x12	slli	0x22		0x32	
0x03	rol	0x13	sll	0x23		0x33	
0x04	flushp	0x14	wrprs	0x24	divu	0x34	break
0x05	ret	0x15		0x25	div	0x35	
0x06	nor	0x16	or	0x26	rdctl	0x36	sync
0x07	mulxuu	0x17	mulxsu	0x27	mul	0x37	
0x08	cmpge	0x18	cmpne	0x28	cmpgeu	0x38	
0x09	bret	0x19		0x29	initi	0x39	sub
0x0A		0x1A	srli	0x2A		0x3A	srai
0x0B	ror	0x1B	srl	0x2B		0x3B	sra
0x0C	flushi	0x1C	nextpc	0x2C		0x3C	
0x0D	jmp	0x1D	callr	0x2D	trap	0x3D	



OPX Encoding

Table 8–2. OPX Encodings for R-Type Instructions (Part 2 of 2)

OPX	Instruction		OPX	Instruction		OPX	Instruction		OPX	Instruction
0x0E	and		0x1E	xor		0x2E	wrctl		0x3E	
0x0F			0x1F	mulxss		0x2F			0x3F	



Assembler Pseudo-Instructions

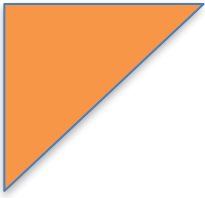
Pseudo-instructions are used in assembly source code like regular assembly instructions. Each pseudo- instruction is implemented at the machine level using an equivalent instruction.

The `movia` pseudo- instruction is the only exception, being implemented with two instructions.

Most pseudo-instructions do not appear in disassembly views of machine code.

Assembler Pseudo-Instructions

Pseudo-Instruction	Equivalent Instruction
bgt rA, rB, label	blt rB, rA, label
bgtu rA, rB, label	bltu rB, rA, label
ble rA, rB, label	bge rB, rA, label
bleu rA, rB, label	bgeu rB, rA, label
cmpgt rC, rA, rB	cmplt rC, rB, rA
cmpgti rB, rA, IMMED	cmpgei rB, rA, (IMMED+1)
cmpgtu rC, rA, rB	cmpltu rC, rB, rA
cmpgtui rB, rA, IMMED	cmpgeui rB, rA, (IMMED+1)
cmple rC, rA, rB	cmpge rC, rB, rA
cmplei rB, rA, IMMED	cmplti rB, rA, (IMMED+1)
cmpleu rC, rA, rB	cmpgeu rC, rB, rA
cmpleui rB, rA, IMMED	cmpltui rB, rA, (IMMED+1)
mov rC, rA	add rC, rA, r0
movhi rB, IMMED	orhi rB, r0, IMMED
movi rB, IMMED	addi, rB, r0, IMMED
movia rB, label	orhi rB, r0, %hiadj(label) addi, rB, r0, %lo(label)
movui rB, IMMED	ori rB, r0, IMMED
nop	add r0, r0, r0
subi rB, rA, IMMED	addi rB, rA, (-IMMED)



add Instruction

Instruction	add
Operation	$rC \leftarrow rA + rB$
Assembler Syntax	add rC, rA, rB
Example	add r6, r7, r8
Description	Calculates the sum of rA and rB. Stores the result in rC. Used for both signed and unsigned addition.



The Devil in the details!

Carry Detection (unsigned operands):

Exceptions Instruction Type Instruction Fields

Following an add operation, a carry out of the MSB can be detected by checking whether the unsigned sum is less than one of the unsigned operands. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition.

The following code shows both cases:

```
add rC, rA, rB # The original add operation
cmpltu rD, rC, rA # rD is written with the carry bit
```

```
add rC, rA, rB # The original add operation
bltu rC, rA, label # Branch if carry generated
```



More Devils!!

Overflow Detection (signed operands):

An overflow is detected when two positives are added and the sum is negative, or when two negatives are added and the sum is positive. The overflow condition can control a conditional branch, as shown in the following code:

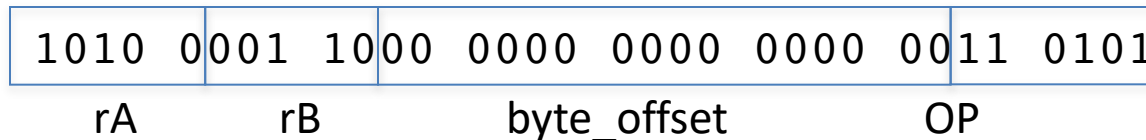
```
add rC, rA, rB # The original add operation
xor rD, rC, rA # Compare signs of sum and rA
xor rE, rC, rB # Compare signs of sum and rB

and rD, rD, rE # Combine comparisons
blt rD, r0,label # Branch if overflow occurred
```

Examples

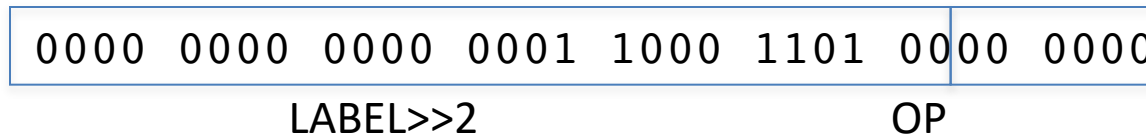
stwio r6,0(r20)

stwio rB, byte_offset(rA)/* this is an I-type instruction */



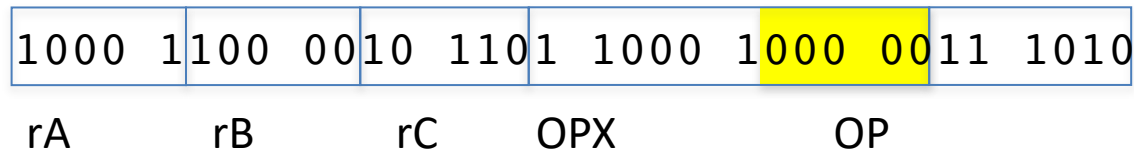
call fred (call is at 0x00000634, fred is 0x000018D0)

call LABEL /* this is a J-type instruction */



add r22, r17, r16

add rC, rA, rB /* this is an R-type instruction */





A Coding example

Create Nios II assembly code to:

Multiply the corresponding elements of two vectors together and then obtain the sum of these values.

Initialise the operations:

- Set up pointers to the vectors

- Set up a loop counter

- Zeroise the sum

The loop:

- Get the values into registers

- Multiply them together

- Add to the sum

- Increment the pointers

- Decrement the loop counter

- If not finished loop again

- otherwise store the result and finish

Example Code



```
.include "nios_macros.s"
```

```
.global _start
```

```
_start:
```

```
    movia r2,AVECTOR
```

```
    movia r3,BVECTOR
```

```
    movia r4,N
```

```
    ldw r4,0(r4)
```

```
    add r5,r0,r0
```

```
LOOP: ldw r6,0(r2)
```

```
    ldw r7,0(r3)
```

```
    mul r8,r6,r7
```

```
    add r5,r5,r8
```

```
    addi r2,r2,4
```

```
    addi r3,r3,4
```

```
    subi r4,r4,1
```

```
    bgt r4,r0,LOOP
```

```
    stw r5,DOT_PRODUCT(r0)
```

```
STOP: br STOP
```

```
N:
```

```
.word 6
```

```
AVECTOR:
```

```
.word 5,3,-6,19,8,12
```

```
BVECTOR:
```

```
.word 2,14,-3,2,-5,36
```

```
DOT_PRODUCT:
```

```
.skip      4
```

```
/*Register r2 is a pointer to vector A*/
```

```
/*Register r3 is a pointer to vector B*/
```

```
/*Register r4 is used as the counter for loop iterations*/
```

```
/*Register r5 is used to accumulate the product*/
```

```
/*Load the next element of vector A into r6*/
```

```
/*Load the next element of vector B into r7*/
```

```
/*Compute the product of next pair of elements into r8*/
```

```
/*Add to the sum into r5*/
```

```
/*Increment the pointer to vector A*/
```

```
/*Increment the pointer to vector B*/
```

```
/*Decrement the counter*/
```


```
/*Loop again if not finished */
```

```
/*Store the result in memory*/
```

```
/*Specify the number of elements*/
```

```
/*Specify the elements of vector A*/
```

```
/*Specify the elements of vector B*/
```



Accessing I/O Registers

In assembler accessing memory with the io option bypasses the cache and accesses I/O correctly:

```
ldwio    r5, 0(r17)                /* load pushbuttons */
```

In C functions are available for accessing I/O:

```
#include <altera_avalon_uart_regs.h>

IOWR(GP_UART_BASE,1,dat);
GP_UART_data=IORD(GP_UART_BASE,0);
```

Alternatively:

/* Declare volatile pointers to I/O registers. This will ensure that the resulting code will bypass the cache*/

```
volatile int * InPort_Key1 = (int *) 0x00009030;
```




Next

A microprocessor that cannot respond to inputs from the outside world and more importantly make changes that affect the outside world will be useless!

In the next section we will look at the hardware and software required in a microprocessor system to read information about the state of the outside world and to output digital signals that can be used to affect the outside world.